Getting faster tex output

Janet Incerpi, Francis Montagnac

INRIA

CENTRE
SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(3) 954 90 20

Rapports Techniques

N° 71

GETTING FASTER TEX OUTPUT

Janet INCERPI
Francis MONTAGNAC

Juillet 1986

Getting Faster Tex Output

Janet Incerpi and Francis Montagnac

Abstract: This paper combines two notes about an output driver for tex. The first, also entitled, "Getting Faster Tex Output," discusses changes and improvements made in an effort to speed up the driver which was running on a workstation. While specific problems about tex output (or dvi) files are addressed, much of what was done applies to any situation where one is trying to optimize code. On workstations where speed is always a concern, a factor of 4 improvement in time is a welcome surprise. Equally pleasing is the fact that we were able to not only have this improvement for the laser printer driver but also for an interactive previewer. Both dvi drivers, workstation and previewer versions, share many modules that process the dvi file. This is due to the fact that we restructured the driver. The results of this work is described in the second note, "Using The DVI Library." The different modules are described as well as how to interface with these to create a new driver, which intends to fill a page of bits and print it. An example of how one such driver runs using this new library is provided. That our goal was a library usable both for output to a bitmap display and a hardcopy device has hopefully improved the design of the interface. With this new format we have found it easy to move the drivers to different machines and Unix systems.

Résumé: Ce rapport combine deux notes concernant le pilote de sortie de tex. La première note, également intitulée "Comment accélérer les sorties de tex," discute les modifications et améliorations qui ont permis d'accélérer ce pilote, utilisé sur un poste de travail. Bien que cette note discute de problèmes spécifiques aux fichiers de sorties de tex (fichiers dvi), bien des observations qui y sont faites concernent la démarche d'optimisation en général. Sur un poste de travail, où la vitesse d'exécution reste un souci permanent, une amélioration du temps d'exécution dans un facteur 4 est une très bonne surprise. Il est également satisfaisant que cette amélioration s'applique aussi bien au serveur d'impression laser qu'à un système interactif d'épreuvage. Les deux pilotes, dans la version serveur laser ou interactif, ont en commun bien des modules de traitement des fichiers dvi. Cette possibilité est la conséquence d'une restructuration du pilote général. Les résultats de ce travail font l'objet de la seconde note, "Manuel d'utilisation de la bibliothèque DVI." Les différents modules qui composent cette bibliothèque ainsi que leur interface d'appel y sont décrits. Cette bibliothèque peut servir de base à tout programme cherchant à composer des pages tex à partir de fichiers dvi. On donne un exemple d'un pilote construit à l'aide de cette bibliothèque. L'objectif d'avoir une seule et même bibliothèque utilisable pour un serveur ou un programme interactif devrait avoir influé positivement sur la conception de son interface. Grâce à cette organisation nouvelle, il nous a été facile de transporter ces pilotes sur plusieurs machines et systèmes différents.

Getting Faster Tex Output

Janet Incerpi and Francis Montagnac
INRIA – Sophia Antipolis

We discuss how to go about improving the speed of an output driver for tex. The driver, which takes as input a tex output (or dvi) file, runs on a workstation. There are two versions: 1) to send output to a laser printer, and 2) to send output to a bitmap display using an interactive previewer. The initial proposal seems reasonable enough, spend a couple days seeing if it is possible to make the driver faster. Clearly this is important for an interactive previewer, where waiting more than a few seconds seems like an eternity. But as well, it is important if the driver is on a workstation.

As is often the case, estimates of the time needed to do a programming job tend to be a little off. It has been said that one should increment the numeric component of the estimate and increase the units by an order of magnitude. That is, if someone says a job will take 3 days then perhaps in 4 weeks it will be completed. This may be due to unexpected problems, or, perhaps, other enhancements made at the same time. Such was the case with the couple days to see if the dvi driver can be made faster. However, a few weeks is worthy investment for such a frequently-used program. Our results were suprising: a speed up by a factor of 4 and a reduction in the space used by 30%.

Here we will talk about the changes we made and the problems we ran into. While we address some specific problems about dvi files, we believe much of the experience is application independent. Thus the story presented here may be useful for people trying to optimize programs on workstations, where speed is a constant cause of concern, as well as others trying to improve dvi drivers.

Our Setup

We begin with a little background information about dvi drivers and our environment. What is a dvi file? It is essentially a number of byte commands. These say to begin a new page, move down, move to the right, place a character, change the font, etc. The driver parses the dvi file and maintains the current position and font for placing each character. Once ready to put a character in the page, the bits that make up the character (the character bitmap) are copied into the page bitmap by a "raster operation." Raster operations are system dependent routines that manipulate bitmaps, in this case taking one bitmap and copying it at some position in another bitmap.

While the changes we made are not machine dependent, in fact, most are even language independent, we mention the environment to complete the setting. The dvi driver is written in the C programming language and runs on unix systems: Sm90 workstations (SMX) and Bull SPS9 (Ridge) computers (ROS Unix). Output is either to bitmap displays attached to both kinds of machines (using an interactive previewer developed by L. Gallot [2]), or to a Canon laser printer attached to an Sm90 workstation.

Optimizing code is not an ad hoc proposition. It is very important to know what, when, and how to optimize. Bentley's "Writing Efficient Programs" [1] is a valuable guide on this subject. As he points out, the first thing to do once deciding to optimize a piece of existing code is to find out where the program spends all its time. We decided to use the basic profiling system that is provided with many C compilers. Using a special compiler option enables information to be gathered for each run of the program. This information includes: how many times a function is called, milliseconds spent per call, total time spent in the function, and the percentage of execution time that this function uses.

Our first delay in "taking a couple days" to improve the dvi driver came in trying to find a working profiling system. This is very important because although one may think it is easy to guess where the time is spent. More often than not the guess is wrong, then all the optimizing in the world will not significantly speed up the code. How could we tell that the profiling system was bugged? We were running the driver on a sample file that produced 8 pages (mathematics and text). This dvi file of over 30000 bytes was read and processed by the driver that was doing all the computation necessary to place over 17000 characters. The profiling system reported that most of the time was spent in the standard C print routine, printf, that was used only to print out each page number as the page was written to the laser printer. Thus, we became suspicious and wanted a second opinion.

While neither computer on which our driver was running had a reliable profiling system, we found a working system on another computer. However, on this machine we could not generate output. We did not see this as a serious problem for two reasons. The first reason was that the raster operations used by the driver are very much system dependent and typically one does not have control over these. The second reason was that by simply timing two versions of the driver, with and without raster operations, we saw that initially the raster operations accounted for only 22% of the total execution time. We were going to focus on that 78% overhead computation necessary to process the dvi file. Thus, settling on a Sun II workstation (running BSD4.2 Unix) we were ready to begin our task. Occasionally we would move the driver back to the Sm90 to test intermediate versions to see how the improvements gained on the Sun fared.

How Time Flies

Now we look at the changes we made to improve the speed, how we came about these changes, and the effect they had on the driver. The first few changes lead to a large initial savings. The latter improvements seem smaller since the program is faster, yet they represent a reasonable percentage of the savings.

Our initial profiling showed that 56% of the time was being spent on floating point arithmetic. Most of this computation was confined to one small, 8 lines including the declarations, routine called pixel_round. What this function does is convert from tex's internal units to pixels ("picture elements"), which are the units of the output device. Tex manages all its computations in "ridiculously small units," we call one such unit an rsu[1]. To position the characters correctly on the page the dvi driver must convert from rsu's to pixels. The conversion was done using a

[1] This name was used in early version of tex, in [3] the units is called "scaled points," denoted sp. The wavelength of visible light is $\approx$ 100sp; thus rsu is still a fitting name.

floating point multiplication, addition, and comparison; this last was necessary for rounding the final result appropriately.

The pixel_round function was clearly the place to start concentrating our efforts. Floating point operations are notoriously slow on machines without special-purpose hardware; be it mainframe or workstation. We wanted an integer version of pixel_round. Of course, we must insure that the results are the same as the floating point version. People want faster tex output but they want it to still look good.

We rewrote pixel_round using only integer arithmetic, including two multiplications. This change lead to a dramatic improvement in the running time, gaining a factor of 2.5. Also our tests showed that this new version had the same results as the floating point version. This due to the fact that we were only outputting to a device with a resolution of 300 pixels per inch. The resolution of a device is its granularity: how many pixel elements are addressable, typically given per inch.

We were to come back to pixel_round again after making all the other changes mentioned below. What happened was that the integer version of pixel_round was sufficiently fast so that new procedures now became suspect. As we worked to speed up these procedures, however, pixel_round was regaining its prominence on the list of big time users. It reached the point where if anything else was to be gained pixel_round would have to be faster still. While integer multiplications are faster than their floating point counterparts, they are, on most machines, still done by software, in the form of an assembly language subroutine. (This is not the case for the Bull SPS9 but for most workstations the hardware works on 16-bits.) Thus, we decided to see if any speed could be gained by making use of short integers, a standard C type that is only 16-bits.

In fact, by carefully examining pixel_round we saw that we could use short integers and have the same accuracy as the integer version for our computations. What pixel_round does is take rsu's, multiply by a "conversion factor", and return pixels. The conversion factor in the original version was a floating point number, now it was broken into two parts and the computation required integer multiplication and logical shifts. That the multiplications could involve short integers means, in the compiler-generated code, replacing two (assembly) subroutine calls with two instructions.

Thus, redoing pixel_round again we saw a good improvement in the running time of the SunII. Note that since the SPS9 is a 32-bit machine we expected to see no improvement. Our wishes were granted, there is no difference between the two integer versions on the SPS9 —it is not often that the predictions one makes are accurate, we were happy to be right! However on the Sm90 we were disappointed: the short integer version was not faster. The Sm90's C compiler had not taken notice that the variables involved were short integers, so it did nothing different. By being a little more obvious, coercing the variables explicitly, the compiler was happy to comply and we got a healthy improvement in speed.

Perhaps making these last changes seems a bit low-level or too drastic but pixel_round represents that "4% of the code that takes 25% of the time" that one often hears about. This makes it the part of the dvi driver that should really be worked over. What we did was not

truly specific to our environment. The module containing pixel_round has a compile time option to get the short integer version (the default being just integers). Thus to move the driver to another system one just tests to see which is fastest for their system. Also the macro facility of C allows us to leave the code clear. No bizarre tricks appear in line, no messy coercions.

While people may readily accept the use of short integers, the most disturbing thing in all this optimizing for tex users will probably be our mentioning of accuracy. Their first question will (and probably should) be: is the computation still correct? Their next question should (and probably won't) be: what is the correct computation? This is what we considered when implementing the new pixel_round. Tex does its computation in rsu's (integers), then the dvi driver computes a floating point conversion factor (based on integers stored in the dvi file), multiplies and converts back to integer units in the form of pixels. The accuracy of our new version depends on the output range of the function. Actually, the horizontal range for the output device is what's important. Theoretically, we can compute the accuracy for standard (A4) paper and a 300 pixel per inch device and show that an off by 1 pixel computation will occur less than .75% of the time (we are never off by more than one pixel). For our sample files the results never differed from the original version. The dvi driver has tests built-in to eliminate accumulated round-off errors because the resolution of the device is probably less than rsu's. We feel that the computation must be rounded at some point, since the output device works in discreet units. Is it correct? Since pixels can't be split and are probably larger than rsu's, the computation must try to be consistent and look good. Only the most highly trained expert can tell a difference of 1 pixel in horizontal positioning at 300 pixels per inch or higher resolutions (at higher resolutions, the pixels are even smaller!).

Something else the profiling revealed was that pixel_round was called over 39000 times for our 8 page example. But as we mentioned above we were placing just over 17000 characters. Twice for each character. Why twice? Initially we thought this was due to the fact that it was called once for horizontal position and once for vertical. However, this is not the case. Generally, the vertical position is updated once for each line of text; only the horizontal position is updated for each character. So, why twice? Upon examining the routine that updates the horizontal position we found that it tested to see what is the appropriate way to do the update. The test required one call to pixel_round and then, no matter the result, another called was made to do the actual updating.

The dvi driver keeps track of the horizontal (and vertical) position both in rsu's and pixels. The routine to move to the right after placing a character checks that the pixel horizontal position hasn't drifted too far from the rsu horizontal position, once converted to pixels. Doing this check requires a call to pixel_round. We found this a costly test. The test was an approximation as to when things seemed a bit off, say more than 3 pixels. Perhaps such an exacting computation is not needed. We tried a test which involved only one multiplication. This new test converts the horizontal position in pixels to rsu's and then compares against the rsu position. The results of our tests never differed from the old test, using a full call to pixel_round, but there are no guarantees that this is the case. Rather we are now resetting the horizontal position, when the drift corresponds to 2.87 pixels, well, that is, if pixels could be split. The point is that the test involves only one multiplication, thus we are calling pixel_round 1.5 times per character. We

saw a sizable improvement in the running time.

That closes the pixel_round story; we made changes to other functions, suprisingly enough, and we turn to those now. Some of our changes were related to the fact that the driver is written in C. Each language has its own "quirks", or pecularities, it is possible to make code more efficient by knowing what these are. For C, these translate into declaring frequently used variables as "register" variables when possible[2], and replacing array indexing with pointer arithmetic. This latter change was very useful in one case. The routine to place each character was passed a font number, by indexing on this value one could find where the font information was stored. This was done for each character! Since more often than not, the previous character is in the same font, we instead passed the pointer to the font information in internal memory. Only when the font changes, do we need to update this pointer.

A more generic change, that also lead to a big improvement, involves always testing for the most frequent case first. The routine responsible for parsing the dvi commands for each page is called, DoAPage, and is one large "while" loop that contains a gigantic switch for each of the 75, or so, byte commands. The code before the switch tested whether the command was a font command, in which case the font was changed, or an ascii character, in which case it simply called the function to place the character. The problem was that it tested for font commands first. It is hard to imagine that the changing of fonts is more frequent than the setting of characters. In the "best" case the font could be changed before each character —here the word "best" refers making good use of the code. Thus we decided to exchange these two cases; inside the while loop we first see if we've got an ascii character. The effect of this change was just about to halve the time spent in DoAPage.

Other gains for DoAPage and the functions it calls came by using macros instead of function calls for some small utility functions. This was used for three functions: Left (to get the left half of a computer word), Right (to get the right half), and GetByte (to get the next byte from the dvi file). These are all replaced with 1 line macros. GetByte was made even faster by turning off the check for end of file (EOF) made at each call. Too risky? Perhaps. But the test for EOF is turned off only for the DoAPage module, thus it is known that we have successfully parsed the beginning and end of the dvi file. Actually, given the way the input/output (IO) works on most systems the check was already redundant most of the time. When told to get one byte from a file, standard IO packages typically a whole bunch of consecutive bytes, 512 or one of those other powers of 2. Thus, only the last byte that is read in the IO buffer must be tested against EOF.

We now saw two possibilities: 1) we could rewrite the IO package to be safer about the EOF check, or 2) we could change the driver to read in all the bytes that make up a dvi page at one time. The first is an interesting idea and it is the type of thing that creeps into such work eventually. It is definitely not the type of change one plans for during the "couple of days". Perhaps we are not brave enough, but we felt that we were reaching the point of diminishing

[2] When a function has many variables none of which dominates, like an important pointer used throughout, then declaring registers may slow things down. However, there are many situations where it works and works well.

return for improvements and were willing to just take the risk —since others may be more cautious these last changes are available through a compiler option. The second possibility causes a problem because it uses more space, storing the dvi commands in an internal array. We hoped to make the driver smaller and this didn't quite fit in with that plan.

How Big Is It?

Besides getting faster tex output, we felt some streamlining could be done. We wanted a slimmer dvi driver, it was quite overweight and working on the workstation which ran the laser printer was less than wonderful. One could notice when something was being printed, that's a problem. The previewer version was smaller and didn't have this problem. Actually the looking at space was partly motivated by the fact that we now wanted both the laser printer driver and the interactive previewer to share dvi processing components.

One of the components to be shared handled the fonts. Basically, the dvi driver just has a large array into which it reads the fonts until it runs out of room, complains, and leaves the user empty-handed. The amount of storage needed depends on the resolution that the fonts were designed for. The fonts needed for the bitmap display are only 118 pixels per inch and thus require much less space. In fact, if the font files were stored without wasting a "bit" of space then the fonts the bitmap display, 118 pixels per inch, would be 6.46 times smaller[3] than for the laser printer, 300 pixels per inch. Even for the laser printer the amount of storage allocated seemed quite large. Our sample file used 10 fonts that filled less than two fifths of the array. But when allocating storage it is not good enough to allocate for the average case since one has to face those who use slightly more storage. To allocate fixed arrays, one must always go for an upper bound. Thus allocating the exact storage needed seems the ideal way to keep the previewer small and quick, and to bring the laser printer driver more in line. Thus we have all the storage needed by the driver allocated dynamicly, including space needed for dvi stacks.

Of course, one reason so much storage is needed is because we have a page bitmap that is filled and then printed. We need space to build a page in memory first. For example, every addressable pixel on the piece of paper for the laser printer requires a bit in memory. Well, maybe not every. What about the margins on the page? Do we need to save place for those bits?? Even so, how much could that possibly save its just the edges of the paper. The edge of the standard (A4) paper, assuming 2.5cm (or 1 inch) of margin all the way around, amounts to $\approx$405000 bytes. Not bad, for just the edges. For the Canon we are able to allocate the page bitmap without any margins since the Canon is able to offset the bitmap we fill on the physical paper. This results, in most cases, in a page bitmap that is 39% smaller. The savings is dependent on the page size the user specifies in their tex file, our figures here are based on a slightly larger page size than the tex default and thus are conservative.

---

[3] Alas, for our system more than a bit of space is wasted. The character bitmaps are padded so that each row is aligned on a computer word (or half word). The end result is that for non-magnified fonts the difference is only 2 times smaller but this increases with the magnification of the fonts up to 4.5 times smaller. This is because the effects of padding are less noticeable when the bitmaps are larger.

The Experience and Problems Along the Way

We now give a quick overview of how time really flies and what kinds of problems and tasks we decided to take up while making the above changes. We also mention a couple of tex problems; actually they are problems with the information tex gives in the dvi file and what we had to do because of it.

As we mentioned above, our work really got off on the wrong foot, by trying to find a system on which we could profile the running time. But this is an irreplaceable tool. We made a rather weak attempt at writing our own little profiler but discovered the tools we needed do not work. Then, the idea to profile on another system came. All became wonderful. Well not right away, the first machine had a bugged profiler as well but by the week's end we had found a system to profile on. Yes, one week down and the driver was no faster yet.

Another important tool, not just when trying to optimize code but for systems work in general, is a revision system. Something that allows different versions of a program to be retrieved without saving multiple versions of the same file (none of this: a, a.old, a.new.old, etc.). The system we used is called RCS, Revision Control System, [4]. Given our situation such a system is a life-saver. We were profiling on a system where we saw no output, on a programs whose objective is nice output! What if we inadvertently did something wrong, the compiler would hardly complain if we accidently deleted a wrong line. With no output to see, we could only make sure that the program would compile and didn't print out that friendly ever-present C message: "Bus error – core dumped." Thus, we would move the RCS archive files back to the Sm90 to see our output (or lack there of).

Of course, revisions systems are not the cure-all that one hopes, they typically have problems handling multiple files. Typically, all the files won't have the same revision numbers and one needs to maintain something to tell which revisions go together to form a working version of the driver. Thus, we maintained a special file named version that would have the RCS identifiers of each module in the driver. This made it easy to find the last version that still produced output and then step through the various versions until all was working. Actually, we didn't have problems of this sort often, but once was enough to find out that a way to incrementally introduce the changes made is a wonderful tool.

In fact, the file management problem we brought upon ourselves in a way. When we first started the work, the driver was one small file to parse user options, such as selected pages, and one large file to do everything else. While it may make handling revision files easy, working conditions when in edit-compile-test mode are less than ideal. We quickly started to separate out certain pieces of code that we were to work with often. However, we soon realized that this was not enough. We decided we wanted something modular, making it possible to share most modules for both versions of the driver we had. Why should making a new dvi driver be difficult? What happens typically is one finds an old driver and hacks at that until output appears. If the output device is happy to receive a page bitmap then so much of the driver should be re-usable. Our goal in breaking up the driver was to separate out the device dependent stuff and provide a clean interface[4]. This also didn't make the driver faster. This also took was another week's

[4] Interested readers should see "Using the DVI Library," intended as a technical note to use

time. That would be week 3 (week 2 was the first integer version of pixel_round, testing, moving back to the Sm90, etc.). Again we felt it was a good investment. Our improvements, first seen for the laser printer, seemed even greater for the interactive previewer while taking very little time to get running.

Besides these more procedural problems, we encountered two technical problems related to the information that tex provides. These involve the page size and page numbers. In the dvi file tex provides the maximum page size, height and width. We were ready to take full advantage of this information by allocating a page bitmap of this size. This is great for the laser printer as we mention above. However, tex doesn't always store the maximum value. In particular, if there is an overfull hbox, whether or not the hfuzz command is used to suppress the tex error message, then these values may be incorrect. Strange that the correct values cannot be stored since it must do some checking to know whether to print an error message! So storing this value should not be too difficult, as it is written at the end of the dvi file. Less obvious are problems that arise because it is possible to convince tex that certain boxes have zero width and then place them in the right margin. For most output, the page size in the dvi file are fine. But most is not good enough, people like to see their output. Our solution, perhaps patch is a better word in this case, is to use the smallest page bitmap possible and add to the driver a user-option to ask for the largest possible bitmap. There is no way for the user to know when this problem will arise, nor can the driver know unless a pass is made through the dvi file doing all the computation to find maximum x and y values. That is not exactly an optimization!

The problem of page numbers is much more amusing. How can the user who only wants to see selected pages access them with the driver? Seems straightforward enough. What's the page number? For the user, it is simply the number that appears at the bottom (or maybe, in the upper righthand corner) on the paper. But for tex and a dvi driver, the page number is just some additional characters to be typeset like the rest of the page. What tex assumes is that a certain internal counter, count0, has an important value, by default, it is the page number. It thus stores this and other counters in the dvi file when it is about to begin a new page. So far, so good. But now suppose the user turns off page numbers (tex, by default, no longer increments this counter), or is using more complex page numbers (e.g., 1–1, 1–2, ..., 2–1, ..., for chapter 1 and chapter 2), or say the user resets the page number since they are texing a few small files at once. In all these cases, there could be multiple pages numbered, say, 3. Assuming count0 is used wisely then how can the driver find, say, the second page numbered 3? What is the most reasonable thing to do for handling such situations?

Our solution (this time, it is not a patch) is to consider that the dvi file consists of, what we call, ranges. A range is a group of consecutively numbered pages. Thus a dvi file that has pages numbered 1 to 10, followed by pages numbered 1 to 5 has two ranges. We are still assuming that count0 has some significance. In the example above with pages numbered using chapter and page number, we hope that count0 is still holding the page number. Ranges make it easy for the driver to access any page selectively. Page 2–2 is the second page of the second range. Or if all the pages happened to be numbered 1 (no page numbers) then the third page is simply the only

---

our dvi driver or its dvi processing components.

page of the third range. Deciding on this structure changed the driver yet again, this work was probably done in week 5 —week 4 must have been the other speed and space improvements.

Although time seems to pass without reason, we were truly happy with the results. No one actually expected that the driver could be made 4 times faster. Of course, the nasty task of writing up documentation doesn't count in the time needed. This time seemed longer than the coding part and wasn't nearly as rewarding!

Results

The concrete numbers appear here, just what is hidden behind all the claims. The tests were done on an 8-page document of text and mathematics (with two small tables thrown in for good measure). The driver spends a small amount of time on overhead processing for the dvi file. This is reading the global information stored at the beginning and end of the dvi file, as well as reading in the fonts. The rest of the time is spent either on processing the dvi information and doing the raster operations to fill the page bitmap.

Before we began, for the laser printer driver, the raster operations that copy the character bitmaps into the page bitmap accounted for only 22% of the 129 CPU seconds used on the Sm90. Now the raster operations take 71% of the 31.4 CPU seconds. We give CPU seconds so that the load of the machine doesn't figure in but these figures are generated when the output is sent to the laser printer. The processing of the dvi information of the 8 pages is 8.9 CPU seconds, if the start-up cost is excluded then for the Sm90 it is just under one (cpu) second per page. As well the space required for this sample file was reduced by 42%. One factor in the amount of space needed is the number of fonts required, our test used 11 fonts.

For the interactive previewer, the real measure of the improvement is how long one sits in front of the screen waiting for the requested page to appear. (Clearly, the CPU time is the same for the dvi processing since we are still on an Sm90.) Real time is important for an interactive program. The "stopwatch" time for displaying a page went from 15 seconds, usable but sometimes annoying, to 4 seconds. How full the pages are effects the time, our tests were on full pages of text. For pages with tables we've seen this time fall to 2 seconds since not many raster operations are done.

The improvements in time actually were different depending on the machine. Different machines, different hardware, different C compilers. In the midst of all our pixel_round testing, we even looked at the assembly language code generated, to see why improvements in timing on the SunII hadn't initially carried over to the Sm90. Below we show the CPU time (in seconds) spent by each of the machines before and after optimizing for our sample file. Note that for these figures no output is generated and no raster operations are performed.

Machine	Before	After
Sm90	100.7	8.9
SunII	35.2	6.8
SPS9	7.0	3.5

Optimizing the driver for the Sm90, the least powerful, of the machines shows the greatest improvements. But even for the SunII, which has a very good C compiler and produces what is

consider good code, the improvement was impressive. Bull's SPS9 is, not suprisingly, the fastest of the three. Perhaps we expected it to be faster, in this case the C compiler can take some of the blame; also the fact that Unix is not native to the machine but is running on top of the Ridge operating system, ROS. However, the numbers are all pleasing to the eye. But the driver is not 10 times faster rather only 4 times faster because the part we improved only accounted for 78% of the original running time. The raster operations and writing the pages now take a greater portion of the time as it should be.

Future Work

Perhaps before closing we should mention a few interesting possible changes to the driver, for those readers with some extra time. What's the next (or best) thing to do? Unclear. As often is the case, there is a conflict between trading space for time. Some changes would improve space but perhaps the gains made optimizing would be lost. One has to decide if that the new improvements are justified: there is maximal gain and minimal loss. Unfortunately, this is comparing apples and oranges.

We first come back to the memory management problem. Parsing the dvi file byte by byte is painful. Reading in, at one time, all of the dvi commands that make up one page is one solution. There is some space lost but compared to the fonts and bitmap perhaps it would not be too bad. However, on a small workstation where space is critical this is not a good use of space.

Fonts are always an issue. They are a possible source of improvement and a probably source of problems. There are some drivers that load only the characters when they are requested. Perhaps this is because only a small amount of space is available; in this case on-demand loading of fonts, as it is called, is a must. If there is no space crunch then the pluses and minuses must be considered. There is space saved. Well, if the font isn't used too much. One has to decide if this is the case. Surely, the main font of the document should be completely loaded. What if there is a lot of mathematics? Of course, maybe only one or two special characters are used from a given font. Is the space saved always worth it? That depends. Reading the font all at once is probably faster than going to do reads every time a character is requested.

Actually it may be that what is needed is better font management. What if reading more than one dvi file. It happens all the time with an interactive previewer and even for a typical driver, one would like to pass more than one dvi file. In these cases, it would be best if the driver knew what fonts were available and did not re-load fonts already present. This requires a mapping scheme since tex associates a number with each font and the dvi file uses to these numbers. The dvi driver has its own internal numbering scheme to reference the fonts. To share the fonts just requires mapping tex font numbers from different files to the same driver internal numbers. For on-demand font loading, a list by internal font number should be maintained and checked against.

With an interactive previewer, ideally, one wants to keep around some "dvi state" information associated with each file. This would allow one to switch between displaying pages from two or more dvi files without processing the global information over and over again. The information that is specific to the dvi file includes things like the conversion factor for pixel_round, the font

names and their corresponding tex font numbers. More storage to keep around this information, but how much more? To run some tests and implement the new changes would probably only take a couple days, three days at the most.

References

[1] Bentley, J. L. *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, N.J. (1982).

[2] Gallot, L. "Ashtex: An Interactive Previewer for TeX," *Proceedings of TeX for Scientific Documentation*, Strasbourg, France, June 1986.

[3] Knuth, D. E. *The TEXbook*, Addison-Wesley, Reading, Mass. (1984).

[4] Tichy, W. F. "Design, Implementation, and Evaluation of a Revision Control System," *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

Using The DVI Library

Janet Incerpi and Francis Montagnac
INRIA – Sophia Antipolis

The dvi library, *dvi.a*, contains the routines necessary for reading and processing a tex output file, also called a dvi file. This note explains how to use dvi.a in constructing new drivers for printing devices or for tex previewers (where the typeset page is viewed on a bitmap). We begin by describing the interface, data structures and functions, necessary for using dvi.a. Next we give an example, in this case, a driver to print tex output on a Canon laser printer. Finally, we describe the library itself, explaining where assumptions about the system are used and different options for creating a version of the library that will run best on various systems.

The Interface

Here we present the interface for dvi.a and a driver. We start by defining the three data structures used. Next we look at how the initialization should be done. Finally, the functions used for printing and handling errors, reported from routines within the dvi library are described.

Data Structures

The first data structure, called *dvidata*, stores information that is needed by dvi.a. It consists of a file pointer to the dvi file, the resolution of the device (in pixels per inch), the full path name to the directory containing the font files, and a user-supplied magnification (called usermag). In a previewing system the user may want to compare pages from different dvi files and thus it is the driver's responsibility to pass a pointer to the dvi file. The magnification changes the fonts used in printing but it does not change the line or page breaks. This is useful if magnified versions of fonts are missing for the device being used. For example, if the math fonts for a bitmap screen don't exist at the magnification specified in the dvi file, the document can be displayed with the same layout using the non-magnified fonts by setting this value appropriately. (See below.)

The second data structure, *dvifileinfo*, holds information read from the dvi file. Besides containing the horizontal and vertical page size, this structure also defines the number of, what we call, ranges. A range is a group of consecutively numbered pages. For example, a dvi file having pages numbered 1 to 10 followed by pages numbered 1 to 5 has two ranges.

The last data structure, called *dvirast*, contains the information about the raster that dvi.a is going fill. It consists of a pointer to the bitmap to be filled, the size of the left and top margin chosen for a page, and the horizontal and vertical size of the raster itself (including the margins). Note that the user may have control over some of this information, perhaps he can change the margins. Also in some situations the raster size could be the same as the size read from the dvi file, or could be taken as the size of the bitmap monitor itself.

All sizes in these data structures are given in pixels. Even the page sizes in dvifileinfo that are based on values stored in the dvi file are converted to pixels by the dvi library.

Functions

Getting the appropriate information from the dvi file is done using the function, DviInitialize. It takes two arguments: pointers to dvidata and dvifileinfo structures. When DviInitialize is called the dvidata structure must already be filled. However, it is DviInitialize which assigns values to the dvifileinfo structure. Note that dvidata's usermag by default should be set to the special constant, DVIMAG, which leaves the magnification read from the dvi file unchanged. To cancel the magnification, usermag should be set to the constant, DVINOMAG. Other possible values for dvidata's usermag change the magnification relatively. That is, +1 sets the magnification one magstep higher, while −1 decreases it by one magstep. However, usermag only effects the overall magnification of the dvi file, it is not possible to effect the magnification of an individual font.

The dvi.a library includes two functions, RastCeiling and RastFloor, that take an integer argument (in pixels) and returns either the ceiling or floor of the argument that satisfies the necessary conditions for raster operations. For example, often the bitmap is assumed to have a width that is a multiple of a half word (or a word) for the computer. Thus, the bitmap may have to be declared slightly larger than the page size. The driver is responsible for allocating the bitmap, thus calling RastCeiling assures that the bitmap has a legal size. DviInitialize also uses RastCeiling when assigning the page size.

While the dvifileinfo structure contains only the number of ranges in the dvi file, it is possible to get the page numbers in any range. This is done using a function called GetDviRange that takes as its first argument a range number and assigns the first and last page numbers in that range to its two other arguments which are pointers to integers. Note that the ranges are numbered starting at 1.

The function, DviWrite, is provided for printing pages from a dvi file. It takes 5 arguments: a range number, the starting page number in the range to be printed, the ending page number in the range to be printed, a pointer to a function which writes the page, and a pointer to a dvirast structure. DviWrite prints each page in the range from the starting page number to the ending page number by filling the bitmap and then calling the write function with the dvirast pointer as the only parameter.

The driver can ask for selected pages from within a range by using the starting and ending page number. DviWrite does the best it can with the information passed to it. Thus, the page numbers requested that overlap those page numbers in the range are printed. That is, if the driver calls DviWrite asking for pages numbered 3 to 7 in a range that contains pages 1 to 5 then pages 3, 4, and 5 are printed. The driver is responsible for doing the error-checking it thinks reasonable on the page numbers. (Note GetDviRange can be used for this purpose.) If DviWrite is called with the starting (resp. ending) page number argument set to the constant DVIENDPT then DviWrite uses the starting (resp. ending) page number for that range. Thus, if the driver simply wants to print all the pages in every range then a call can be made to DviWrite for each range number with the page number arguments set to DVIENDPT. DviWrite returns a code that indicates errors when there are problems with the starting and ending page numbers or the range number. These codes are DVISUCCESS (at least one page was printed from the

range), DVIBADRANGE (range number is bad), DVINOPAGES (no pages requested exist in the range), DVIPAGEBADSIGN (requested positive page numbers in a range with negative numbered pages, or vice versa), and DVIBADFROMTO (starting page number is greater than ending page number, or the page numbers differ in sign). Page numbers within any range are always of the same sign. Note that for tex negative page numbers are not treated as their integer values but rather they are treated the same as their positive counterparts. That is, the page numbers are incremented: -1, -2, etc. and the negative sign simply tells tex that the page number should be printed in roman numerals. For dvi.a, negative page numbers are again not viewed as arithmetic values. A range can contain pages -1 to -5 and the starting value for this range is -1 since in the dvi file page -1 is followed by the page numbered -2. (Pages numbered zero appear in ranges with positive page numbers.)

DviWrite uses the information in the dvirast structure, so the driver must initialize this before the call to DviWrite. The driver allocates the storage for the bitmap and places a pointer to it in the dvirast structure. The top and left margins tell where in the bitmap dvi.a starts placing characters. It defines, in the bitmap to be printed, an origin (or offset) for the page read from the dvi file. The raster sizes in the dvirast structure are used by the library routines that place characters. In particular, given the horizontal position for the character and its width if the bits for the character would extend past the width of the raster only that portion of the character positioned within the width of the raster is printed. The same holds for the height of the characters as well. Essentially the dvi library routines do raster operations to move a character bitmap into the page bitmap. Once the page bitmap has been completed, DviWrite calls the write function argument with the dvirast pointer. The driver must define this write function that can use the raster size information from dvirast. Thus, for a previewer the write function might simply be a raster operation of the page bitmap onto the screen.

Finally, the driver must define a function to handle errors that are detected by the dvi library. This function named DviMsg takes two arguments, the first is a severity code (an integer) and the second is a pointer to a character string containing the message to be displayed. This should be handled by the driver since a previewing system probably won't use simple print statements to display the messages. The severity code indicates whether the message is a DVIERROR, usually fatal, a DVIWARNING, something is amiss but we continue, or a DVICOMMENT, information that need not be displayed in all systems.

An Example

We now describe a driver that uses dvi.a to send output to a Canon laser printer. We found it convenient to split the driver into two parts. One part containing all the device-dependent routines; the other containing some initializations for defaults, the parsing of user options, and appropriate calls between the device routines and dvi.a. The reason for dividing the driver in this manner was the hope that other tex drivers could make use of the latter part. Both parts of the driver use the dvi library include file that defines the data structures and error codes described above.

There are three device-dependent routines that we created: DeviceResolution, DeviceInitialize, and WritePage. DeviceResolution simply returns the resolution of the laser printer.

DeviceInitialize takes a pointer dvirast structure and a flag as arguments. The dvirast structure has the top and left margins set and the raster sizes containing values provided by DviInitialize. The reason for initializing dvirast in this way is that the Canon uses a special ioctl function call to set its own margins and raster size. With the Canon it is not necessary for the bitmap to contain the zero bits that comprise the margins. The write routine for the laser printer will offset the raster on the piece of paper according to the top and left margins specified in the ioctl call. Thus, our raster size can be exactly the page size read from the dvi file. Since the Canon handles the margins, DeviceInitialize resets the margin values in the dvirast structure to zero after doing the ioctl call.

Also DeviceInitialize allocates the bitmap and stores a pointer to the bitmap in the dvirast structure. Note that the Canon's handling of margins is very special. If it did not manage the top and left margins then our driver would assign to dvirast the nonzero values the user wanted for margins; for the raster sizes we would assign the page size from dvifileinfo plus the margin. This is because the raster would now contain the bits for the margin, dvi.a would start placing characters using the margins as offsets. DeviceInitialize's second argument, the flag, is simply an indication that the user wants the largest possible raster, by default we use the information in dvirast to generate the raster. This option is necessary since *in tex it is possible to have output that is wider than what the dvi file claims is the maximum page width.* Finally, DeviceInitialize does some error-checking that insures that the raster horizontal and vertical size plus the margins are not too large for the laser printer and returns an integer error code.

The function WritePage sends a page of output to the laser printer. This function takes as argument a pointer to a dvirast structure. (This is necessary since it is passed as the write function to DviWrite.) It also does checking for errors such as no more paper or paper stuck in the printer.

The other part of the driver, first, sets some default values and parses the user's options. Next it initializes the dvidata structure. This requires setting a file pointer to stdin and the resolution. After we call DviInitialize with pointers to the dvidata and dvifileinfo structures. The driver uses the information returned in dvifileinfo to assign values to the dvirast structure. Once this has occurred DeviceInitialize is called. The dvirast structure, which is modified by DeviceInitialize, and the WritePage function are among the arguments when DviWrite is finally called to generate the requested output.

We now briefly describe the user options available and the defaults that the driver uses. There are some options that restrict output such as choosing a starting page number, an ending page number, and a range number. Other options change the page's appearance like setting the magnification, the left margin, and the top margin. There are also two options that help to use different fonts with the driver. One sets the resolution and the other gives a path name to the directory to search for fonts. Finally there is an option to ask for the largest possible raster for the Canon.

The defaults are such that no page numbers or range number means all the pages in all the ranges. The user can get all the pages of one range by just specifying the range number and no page numbers. Requesting from page 3 to page 5 without a range number results in

all pages numbered 3 to 5 to be printed from every range. Note that our driver does not use GetDviRange to find out the page numbers corresponding to any range number. However, it may be that a previewer would like to display the page numbers when the user asks to select a page. GetDviRange would be useful for this purpose.

The default magnification is DVIMAG (the magnification value stored in the dvi file). This value can be changed using a signed real number indicating increases or decreases by magsteps, or by specifying zero to indicate no magnification is desired. In this last case the driver sets dvidata's usermag field to DVINOMAG. The margins are specified in centimeters, the default is 2.5 centimeters for both top and left margins. While these may be good defaults for the laser printer, for a bitmap this white space seems unnecessary.

The resolution by default is set using the DeviceResolution function. But the user may specify some other value (in pixels, of course). This may be useful if the fonts available do not match the resolution of the output device. The resolution is used by dvi.a both for computing the font names (the extension is computed using the resolution and the magnification) and for computing where to place the characters on the page. The path name to the directory containing the font files is, by default, set to the value of the shell variable TEXPIXELS. But an option is provided to change the path name by resetting the dvidata field, fontpath.

The last option for the largest raster possible needs a bit of explanation. Our driver by default tries to use the smallest raster possible and thus it uses the information read from the dvi file to know the size of the page. However, it is possible in tex to generate lines that are longer than what the tex file claims is the width of the page and for tex not to give an overfull hbox error. This option is used solely to get around this problem that tex does **not** store in the dvi file the **true** maximum width for the pages.

The final procedure defined by the driver is DviMsg. This is the error message routine that is used by dvi.a. Recall this function takes two arguments a severity (integer) code and a pointer to a string that is the message to be printed. Our driver prints all messages using a simple print on stderr. However, a previewer would probably like to open a special window to display errors or warnings when they occur.

The Library

We now describe dvi.a itself. We begin with the various components. Next we talk about where assumptions are made about the system and device, and how to get dvi.a to work on a different system. Finally, we talk about optimizations that we have done and the compile time options that can help produce the best dvi.a to be used on a given system.

The Components

The dvi library consists of 6 components: dvimain, dvifonts, dvipage, dvirast, dvipixel, and dviutils. There are also three include files that are used by various modules of dvi.a.

One include file, dvidriver.h, contains the definitions for the dvidata, dvifileinfo, and dvirast data structures. Also it contains constants returned by the dvi modules. These include DVIFAIL and DVISUCCESS, as well as errors detected by DviWrite such as bad range number, no pages with numbers requested in the range, etc. Also the severity codes for DviMsg and the constants

DVIENDPT, DVIMAG, and DVINOMAG are defined here. This include file is needed by the driver as well as all the dvi library components. Another include file, dvi.h, contains the integer command codes that make-up the dvi file. Only those components that actually interpret what is read from the dvi file need dvi.h. There are three such components: dvifonts, dvimain, and dvipage. The final include file, dvimain.h, contains the data structures, constants, macros, and global variables that must be shared between different components of the dvi library but are not needed by the driver.

The dvimain module contains the DviInitialize routine and all functions that are called to get global information from the dvi file. (Information read from the preamble and postamble of the dvi file.)

The dvifonts module contains functions for generating the font names, reading the fonts, and accessing them as well. The storage for the fonts is allocated dynamically. This is best since the amount of space needed would be dependent on the number of fonts and size of the fonts. For example, the font files used for a 300 dots per inch (dpi) device are larger than those needed by a 118 dpi device. However, this dynamic allocation is sometimes a problem for systems that cannot dynamically change the stack size. Setting the stack to 9K seems adequate unless one is using a large number of fonts.

The dvipage module contains DviWrite and functions that help to parse the dvi commands that constitute a page of output. The dvi commands say to start a page, change a font, move to the right, move down, place a character, etc. However, the actual placing of the bitmap of the character into the raster is done by the dvirast module. This module is separate because raster operations are clearly very system-dependent. The functions RastCeiling and RastFloor are two functions that exist in this module. As well, there are functions for clearing the raster and performing the necessary initialization.

The dvipixel module contains a few very small functions that are separate because they are very important. Tex has its own internal units that it uses to measure distances and sizes. These are called rsu's (for "ridiculously small units"). However, when a character is to be positioned in the raster the unit of measure is pixels (dependent on the resolution of the fonts or output device). Thus to position any character or to move any distance a conversion from rsu's to pixels must be made. The functions in dvipixel are all involved in the conversion or checking for accumulated errors that can occur from working with pixels instead of rsu's. These functions are important since the dvi.a spends a lot of time in the routines here, in particular in a function called pixel_round. This function takes an integer in rsu's and returns the rounded value in pixels.

The dviutils module contains several small functions that are used to get bytes from the dvi file, go to a particular position within the dvi file, and to extract information that is packed in the dvi file (e.g., read the left half of a computer word). These are utilities needed to properly interpret information that is stored in the dvi file.

Assumptions Made

The dvi library is written in the C programming language. It is designed to run on a Unix system. We implemented it on a SM90 workstation as well as a BULL SPS9 (Ridge) computer.

The different modules were created to aid porting the library to different systems and using it with new output devices. Here we mention assumptions that are made and what must be done to get dvi.a running on a different system.

There were some assumptions made about the dvi file: 1) the dvi file is a version two file, and 2) the page numbers are taken to be the values stored along with the beginning of page command in the dvi file (traditionally, tex assumes that its internal counter "count0" contains the page number). Clearly, these assumptions depend on tex and the user. If what tex outputs is changed then the library must be modified, however, these assumptions are not a problem when using the library with tex82 output.

Since the dvi file contains its information packed as bytes, getting information from the dvi file may be a source of problems when porting to a new system. These routines are contained in the dviutils module. They include things such as moving to a particular place in the dvi file via "fseeks" both from the beginning of the file (GotoByte) and from the current position (GotoRelByte). Also there are routines to get the parameters to dvi commands by recombining 1, 2, or 3 bytes of the dvi file into the correct integer value. For the GetByte, Get2Byte, and Get3Byte functions the problem is getting the sign extension correct. The bytes are read one at a time and combined using logical or and shifts. To get the sign extension when combining 2 or 3 bytes, a short integer containing the leftmost two bytes is assigned to an integer variable. The system is assumed to do the correct conversion (another shift and or gets the third byte).

Another place where assumptions about the system are important is in the dvirast module. This module does the raster operations and also contains the RastCeiling and RastFloor functions. For a new system one must add to the dvirast module the appropriate #ifdef to access the necessary include files for the data structures. Also #ifdef's must be added to the DviRastInit, ClearRaster, CharSet, and RuleSet routines. Note this last function is used to draw horizontal and vertical lines, tex does not have characters for these lines (or rules as they are called) rather it tells the length and position and the dvi library (through raster operations) creates them. The RastFloor and RastCeiling function may also require additional #ifdef's since it adjusts the pixel values (up or down) to be a multiple of, say, 16 or 32 or whatever is necessary for the raster operations. (Recall the dvifileinfo page sizes adjusted by dvi.a using RastCeiling.)

There are also implicit assumptions in the dvipixel module. The computations used in pixel_round to convert from rsu's to pixels are done using integer arithmetic, this replaces an earlier floating point version. Whether pixel_round is acceptable depends on the resolution of the output device. The range of the pixel_round function determines if the accuracy is acceptable. That is, if in less than 1 percent of the cases pixel_round returns a value that differs from the floating point version by 1 pixel can this be seen on a bitmap display? The dvi library has three integer versions of pixel_round: one using short integers, one using unsigned short integers, and one using integers. The accuracy of using signed shorts is half that of the other two. For both a bitmap and a Canon page, there is no problem with the integer arithmetic. For devices with much higher resolution, say 3000dpi, for a standard page perhaps 10 percent of the time the computations are off by 1 pixel, however, one pixel is 1/3000th of an inch! Whether this is acceptable aesthetically is another matter. Output devices work in discrete units so even with

floating point the value returned by pixel_round is simply rounded up to the nearest pixel.

To move dvi.a to another system care should be taken about the type conversion between the various kinds of integers. As mentioned above, this is necessary for sign extension. The library assumes that a short integer is 16 bits and this is half the size of an integer. However, for most systems these assumptions are not a problem.

Optimizations and How To Compile

We have optimized the dvi library, the speed increase is important both for the bitmap (used with a previewer) and for the laser printer. The most dramatic speed up comes from changes to the dvipixel module. Removing the floating point arithmetic is very important for workstations without special hardware. Also the improvements take advantage of the instruction set of the machine (using ints, shorts, or unsigned shorts), this to get the maximum improvement. For example, there is no advantage on the BULL SPS9 (Ridge) to use shorts as it is a 32-bit machine. For each machine one should decide what works best then compile the dvipixel module with the appropriate flag to get the fastest version. The compiler flags are named MULS (to use short integers) and MULU (to use unsigned shorts), with the default being integer multiplication.

Another time consuming routine is DoAPage which is in the dvipage module. This loops through a giant switch getting bytes and parsing the dvi commands that make up the page. Since when this is called one has successfully parsed the postamble and preamble we felt some speed could be gained by not checking for end of file each time a byte is read from the dvi file. Also we use pointers to short integers (rather than logical AND, etc.) to access halfwords in the font information. These changes are implemented and can be accessed using the MACRO compiler option, which only affects the dvipage module.

To compile the library there are also a couple other options which may be helpful if one is trying to implement a new driver or trying to test some portion of the dvi library. These flags are OUTPUT (to have some), RASTER (to actually do the raster operations), and DEBUG (to print out some debugging code). These flags turned out to be quite useful when trying out new optimizations or options. The OUTPUT and RASTER flags allowed us to profile the library without outputting to see just the time spent by the dvi.a routines. The OUTPUT flag being used for allocating the page bitmap and opening the Canon, as well as, in the dvirast module indicating the correct include files or defining the bare necessities. The RASTER flag is only used in the dvirast module. If not defined when compiled no raster operations are done and the routines that are normally supposed to be defined in a library are defined as empty stubs. When adding a new #ifdef for a new system the stubs should be added within the #ifndef RASTER in dvirast. The DEBUG flag is perhaps most useful. Throughout the code there are various "show(var,fmt)" statements. This macro is defined in dvidriver.h and when DEBUG is on it prints the variable, var, to stderr using the format, fmt, supplied, otherwise it is replaced with the empty statement. For example, "show(size, %d in DviWrite)" results in "size = 45 in DviWrite" to be printed.

To compile the library one need only determine the CFLAGS that must be set in the makefile. Decide what works best for the machine: MULS or MULU or nothing. Also decide if the savings from MACRO is worth it (we think so). If a new system variable such as SUN

has been added to the dvirast module then this must be a flag as well. Both OUTPUT and RASTER must be on if anything is to be printed or displayed. If running on a new system the raster operations may require a specific library be loaded and thus in the makefile the value of LIB should be adjusted. Finally, make sure no modules have been compiled with DEBUG once ready to create a new dvi.a since this output would not be of interest to anyone but those people who have to get a new driver working. Note that while most of the #ifdef's are only used in one or two modules (DEBUG is the exception) just setting CFLAGS in the makefile once and for all works best.

Hopefully, the only real changes to the library occur within the dvirast module for implementing on a new system. By limiting the changes to these routines and the writing of the device-dependent driver, dvi.a may be a helpul to others.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique